ESE 530: Computer-Aided Design

Project 3: Design of CNN architecture under constraints onto a Nvidia Kepler GPU Architecture

Likhith Harish Anumukonda - 114963244
State University of New York
Stony Brook, NY, USA

Vinith Kumar Pasula - 114931500
State University of New York
Stony Brook, NY, USA

Table of Contents:

1. Introduction

The CH-CNN algorithm from the paper[1] is a method for optimizing CNN architectures using an evolutionary strategy. It is designed to evolve over generations to find a CNN that is accurate and satisfies certain resource constraints such as the number of parameters. The CH-CNN framework begins with initializing the population of CNN architectures using a predefined encoding strategy. The population evolves over a set number of generations, during which the fitness of each CNN is evaluated, penalized if necessary, and possibly repaired to meet resource constraints. Genetic operators such as crossover and mutation are applied to create offspring, which are then selected based on their fitness to form the next generation. This process continues until the maximum number of generations is reached. The initial population is created by randomly generating individuals that encode CNN architectures. Each individual's architecture is represented by a sequence of building blocks and pooling layers. These blocks include various types of convolutional layers, skip connections and group convolutions. The length of each individual represents the depth of the CNN and is determined by a random integer within a specified range.

The scope of this project aims to map the CNN blocks defined in the population initialization phase onto a true computing architecture such as the Nvidia Kepler GPU or XILINX FPGA board. The four CNN blocks namely:

1. **skipV0**: The skipV0 block includes two convolutional layers, each with a 3x3 kernel size. The feature map sizes are denoted as "64-d" for the input and "64" for the output, where "d" likely represents the depth of the feature maps. The skip connection bypasses the two convolutional layers and adds the input directly to the output of the second convolutional layer. This addition operation is depicted by the "+" sign, and it serves to mitigate the vanishing gradient problem and allows the training of deeper networks.
2. **skipV1**: This block is a variant of the skipV0 block that incorporates bottleneck layers to reduce the number of parameters. The bottleneck structure usually involves a 1x1 convolution to reduce the depth, followed by a 3x3 convolution, and another 1x1 convolution to restore the depth. The skip connection here functions similarly to that in the skipV0 block.
3. **groupV0**: This block illustrates a CNN block that uses grouped convolutions. Grouped convolutions divide the input feature maps into groups and perform convolutions within those groups. This reduces the number of connections and parameters compared to a standard convolution. In the figure, the "group = 4" indicates that the feature maps are divided into four groups for convolution. The block diagram shows a 3x3 convolution, suggesting spatial convolutions are performed on each group.
4. **groupV1**: Similar to the groupV0 block, but with a bottleneck architecture applied to the group convolutions. This block is likely a more parameter-efficient version of the groupV0 block. It includes a depthwise separable convolution, which is a form of grouped convolution where each group has only one channel. This is followed by a pointwise convolution (a 1x1 convolution) to combine the outputs of the depthwise convolution. This structure significantly reduces the number of parameters while maintaining representational capacity.

Figure 4 from the paper [1] depicts an example of a CNN architecture and its corresponding genotype encoding. The architecture consists of an input layer, several convolutional blocks (denoted as block1,

block2, block3), a mean pooling layer, and an output layer. The genotype encoding represents the configuration of each layer in the architecture, where:

- "32-64" represents block1 with 32 input and 64 output feature maps.
- "64-256" represents block2 with 64 input and 256 output feature maps.
- "0.8" represents a mean pooling layer. The number between 0 and 0.5 would indicate max pooling, while a number between 0.5 and 1 indicates mean pooling.
- "256-512" represents block3 with 256 input and 512 output feature maps.

Each of these blocks corresponds to the building blocks described in the algorithm, where different types of blocks (convolution, skip, group) can be combined to form a diverse population of CNN architectures for the evolutionary process.

Nvidia's Kepler family GPU architecture provides many techniques for optimizing the performance of the CNN blocks such as the cuCDNN library for faster convolution calculations, Layer fusion, Memory management, asynchronous execution, kernel Optimization, etc. One challenge that arises using a Kepler GPU is that the CUDA libraries do not support grouped convolution which is one of the major steps in the algorithm that is aimed at reducing the number of parameters. Hence this would require us to implement that using custom CUDA kernels.

We have considered the architecture of the Nvidia Kepler K40 GPU in designing the schema for mapping the CH-CNN blocks. The NVIDIA K40 GPU has 2880 CUDA cores in 15 SMX units and 12 GB of GDDR5 memory. Understanding this will allow us to design CNNs that can fully utilize the parallel processing capabilities without exceeding memory limits. For instance, we can ensure that the number of threads per block is a multiple of 32 (the warp size) for optimal performance.

We have also considered two approaches from recent papers on CNN mapping to Kepler GPUs [2][3] in optimizing our solution. The paper, 'Optimizing Memory Efficiency for Convolution Kernels on Kepler GPUs' [2] presents optimizing kernels for a special case on single input channels and a general case for multiple channels. Since our problem falls into multiple channels, we have considered using a block partitioning strategy and a 2D thread block layout to facilitate data sharing and reduce memory communication as mentioned in the paper. The second paper, 'Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs' [3] presents several optimization approaches like L1 cache bypassing and kernel fusion to enhance performance.

2. Approach to Mapping CNN on Kepler GPU

2.1 NVIDIA Kepler K40(Tesla K40) Architecture:

We have considered mapping the CH-CNN algorithm onto the Kepler K40 GPU. The Nvidia Kepler architecture, exemplified by the Tesla K40 GPU, is a significant development in GPU design, providing substantial improvements in performance and energy efficiency compared to its predecessors.

1. **Streaming Multiprocessors (SMX):** The core of the Kepler architecture is the SMX, this provides higher performance per watt than previous generations which are optimized for parallel processing. The K40 has 15 SMX units, each with its own L1 cache/shared memory and a set of registers.
2. **CUDA Cores:** The K40 has 2880 CUDA cores, offers massively parallel processing power for high-throughput computing tasks, and is organized into multiple SMX units, allowing them to execute thousands of threads simultaneously.
3. **Memory Hierarchy:** Efficient use of the memory types is critical for achieving high performance on the Kepler K40 GPU. It optimizes memory usage by reducing latency, increasing bandwidth utilization, and maximizing computational throughput.
    A. **Global Memory (GDDR5):** Largest and main memory pool accessible by all threads,12 GB capacity in the Kepler K40, suitable for large datasets, Slower access times but essential for large-scale processing tasks.
    B. **L1 Cache and Shared Memory:** Configuration: In the Kepler architecture, L1 cache and shared memory are configurable. The total available on-chip memory per SMX (streaming multiprocessor) can be split between the L1 cache and shared memory based on the needs of the application. Size: Typically, it can be configured in a 16 KB/48 KB or 48 KB/16 KB split for L1 cache/shared memory respectively.
    C. **Registers:** The fastest memory available to CUDA cores. Used for thread-specific variables, offering very low latency. Per Thread: Each thread in a warp has access to a set of registers. The K40 has 64 KB of register file size per SMX. Speed: Registers offer the lowest latency access compared to other memory types.
    D. **L2 Cache:** Size: The Kepler K40 features a unified L2 cache that can be up to 1.5 MB. Function: This cache serves as a bridge between the high-speed registers/L1 cache and the slower global memory, reducing the need for frequent global memory access.
    E. **Constant Memory:** On-chip cache for variables that do not change during kernel execution.Best for data that is read by all threads and does not change over time.
    F. **Local Memory:** Used when registers are exhausted. Stored in global memory, so it has high latency and low bandwidth.
4. **Dynamic Parallelism:** This feature allows GPU threads to dynamically spawn new threads, enabling more complex and variable-structured computations to be efficiently executed on the GPU without going back to the CPU for more instructions.

5. **ECC Memory Support:** The K40 provides optional Error Correcting Code (ECC) memory protection for its internal memories, which is critical for scientific and high-precision computing tasks where data corruption cannot be tolerated.

2.2 Theoretical concepts utilized for implementation:

1. **GPU Memory Constraints and Modeling:** The common challenge in GPUs is the utilization of memory bandwidth. The Shared memory Bank width (which is 8 bytes on Kepler G40) should match the Computation data width for higher performance.Let W (SMB) be the Shared memory Bank width and W(CD) Computation data width for each thread and the relation between them is W(SMB) = n*W(CD).As W(SMB) is 8 for Kepler G40 if each thread accesses 2 elements together in a single access for a float (WCD = 4) as a unit then the Shared memory Bank width and W(CD) Computation data width are matched resulting in a 2 times improvement in SM bandwidth. Similarly for the algorithm to perform better the access patterns along with computation patterns need to be designed such that we get a n times improvement.
2. **Data Sharing in Convolution:** Apart from matching the bank width, it is also important to maximize data sharing which helps in reducing memory access and in turn memory communication. For example, consider general convolution on the image. The same pixels can be reused in both horizontal and vertical directions highlighting the data reuse method. So, the memory access patterns, and the computation patterns need to be designed/used while adhering to the constraints of the GPU memory hierarchy. This helps memory reuse and reduce both the global memory (GM) and the shared memory (SM) accesses.
3. **Parallelism in Building Blocks:** Parallelism plays a major role in achieving high performance for CNN on GPU, especially for Kepler K40. The most common level of parallelism on GPUs is at the thread level where each thread computes the convolution operation for a specific pixel or set of pixels in parallel and is stored in registers or Shared memory for further processing. Also, organizing these threads into blocks and executing these building blocks in parallel is critical for optimizing performance. So in the big picture, the threads are organized to form building blocks and indeed the building blocks are organized to form convolution kernels allowing for kernel-level parallelism where different convolution kernels are executed concurrently in GPU. The building blocks of K40 GPU CNN such as skipV0,skipV1,groupV0, and groupV1 use the above Parallelism techniques to optimize performance.

2.3 Complete Implementation of CH-CNN algorithm on Kepler K40

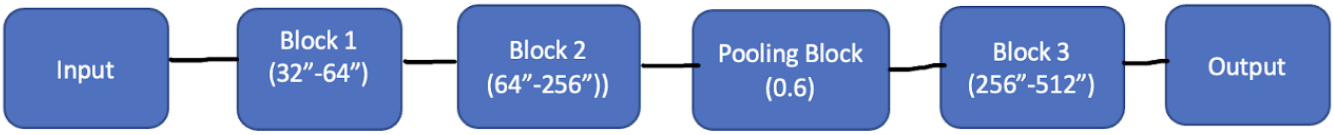
- **Population Initialization:** The Kepler K40 GPU initializes by allocating memory for the population of CNN architectures. Each CNN genotype is stored in the global memory of the GPU, which has high capacity but relatively slow access speeds.Kernel functions for random number generation are called to create the initial population based on the encoding strategy.
- **Fitness Evaluation and Penalty:** For each architecture in the population it determines the optimal batch size for training based on Architecture and trains each individual CNN. In the case of Kepler K40 with 12GB on-chip memory and considering the CIFAR-10 dataset, if each image corresponds to a size of 5MB, a batch size of 1000 images can be efficient. This would allocate memory for other operations accessing global memory. After that, it evaluates the

fitness/performance based on accuracy and constraint violation and stores them in Registers (The architectures are still in Global memory for future reference)For each CNN, the training process is executed in parallel. The K40's SMX (Streaming Multiprocessor) units allow for many threads to execute concurrently, with each thread handling the forward and backward pass computations of different parts of the network. Shared memory, which is faster than global memory but has a limited size, is used to store intermediate data such as feature maps and weights during computation.

- **Population repair:** For each architecture in the L1(or L2) cache check if there are any memory limits (or any other resource violations) and if so then put them in the L2 (or L1) cache and repair them to reduce the size (or based on resource requirement) to make it fit for the resource constraints and make them feasible.The goal here is to find a model closest to the constraint value, ensuring it's within the feasible range. If no suitable model is found, the original infeasible model is retained and stored back in L1. This repair method enhances the feasible, efficient CNN architectures list and stores them in the L2 cache for further processing. A kernel function evaluates the complexity of each CNN and applies a penalty to those exceeding the constraints. This penalty is computed in parallel across all individuals in the population. An adaptive repair mechanism adjusts the CNNs to bring their complexity within the defined constraints. The repair operations are also parallelized, with each thread modifying a different part of the CNN's structure.
- **Offspring Generation:** Crossover and mutation are implemented using parallel kernel functions. For crossover, threads are paired and exchange parts of the CNN genotypes. For mutation, individual threads introduce random changes to the genotypes. These operations can be performed in-place in the global memory or use additional buffer space as needed.
- **Environmental Selection:** A parallelized roulette wheel selection kernel function is used to probabilistically choose individuals based on their fitness. Elitism is enforced by copying the best-performing individual's genotype into the next generation, which can be done efficiently using a parallel reduction to find the maximum fitness value and cudaMemcpy functions to transfer data.

2.4 Example of an encoded CNN Architecture mapped onto K40 GPU

- **Input Data Preparation:** The input image data is loaded into the global memory of the GPU. Preprocessing kernels are launched to normalize the data and prepare it for the CNN.
- **CNN Processing:** The encoded CNN from the below figure (e.g., "(1-32-64)-(2-64-256)-0.6-(3-256-512)") is decoded, and its architecture is constructed in memory. Each convolutional block is set up to be processed by a different set of threads. For the mean pooling layer (indicated by "0.6"), a kernel function is called where threads compute the average of the input feature maps.
- **Forward Pass:** The forward pass of the CNN is executed with each layer's computation distributed across many threads. Convolutional operations are highly parallelizable, so the GPU's CUDA cores are utilized efficiently here. The result of the forward pass is stored in global memory, which will be used during backpropagation for gradient calculation.



Complete flow for the above example:

- **Input Block:** The input data is initially loaded from the 12GB GDDR5 global memory, which serves as the main storage area for data that is accessed across various GPU kernels.
- **Block 1:** Kernel configurations allow this block to process 32 input channels and output 64 feature maps. It utilizes a combination of SkipV0 and GroupV0 blocks for parallel processing. GroupV0 reduces parameters by partitioning feature maps into groups, thus requiring fewer resources and allowing for memory access patterns that match the GPU's warp size. The SkipV0 block performs additional convolutions, leveraging the shared L1 cache to store intermediate feature maps. This shared memory is faster than global memory, reducing latency during data access within the block.
- **Block 2:** Transitioning from 64 to 256 feature maps, this block combines the efficiency of GroupV1 with standard convolutions. It accesses the outputs stored in the L1 cache from Block 1, using this shared memory to avoid frequent global memory reads and writes. Registers—a type of on-chip memory—are employed within the block for storing temporary, frequently accessed data, which helps decrease memory traffic and improve computation speed.
- **Pooling Block:** The pooling layer simplifies the feature maps using an operation signified by "0.6," a mean pooling variant. This process consolidates data from the local memory cache, which is part of the global memory but designated for the current block's use. By strategically pooling in the local memory space and utilizing shared memory for intermediate storage, the block maximizes data reuse and minimizes latency.
- **Block 3:** For the final convolution layers, prefetched data from local memory is transferred back to shared memory, ensuring that the data is readily available for the next set of operations. Block 3 employs bottleneck structures from SkipV1 and efficient group convolutions from GroupV1. The output of these convolutions is stored in the larger L2 cache, which serves as an intermediary storage area before the data is returned to global memory or passed to the next layer.
- **Output Block:** The completed feature maps with reduced parameters are stored back in the global memory. The global memory holds the different CNN architectures for evaluation, acting as the repository from which data is distributed and gathered from the various blocks.

Throughout this process, the GPU's memory hierarchy is strategically leveraged:

- **Global Memory:** Used for storing large datasets and CNN architectures that are accessed by all blocks.
- **Shared Memory (L1 and L2 caches):** Acts as a low-latency buffer for data that's shared within blocks or between certain stages of the pipeline, facilitating fast access to intermediate results.
- **Registers:** Utilized within blocks for the fastest access to temporary data during computation.

By coordinating memory access and sharing across these types, the architecture is designed to minimize data movement and access times, enhancing the overall efficiency and performance of the CNN on the K40 GPU.

2.5 Mapping the Building blocks of the CNN onto the K40 GPU

Skip V0, Skip V1, Group V0, and GroupV1 blocks of Kepler K40 Architecture:

Mapping the CNN blocks onto the Nvidia Kepler K40 GPU architecture involves understanding and leveraging the architectural features and memory hierarchy of the GPU to optimize the execution of various neural network layers.

1. **skipV0 Block:** Utilizes two 3x3 convolutional layers with a skip connection. The implementation would employ shared memory and L1 Cache for the kernel data to optimize convolution operations. The skip connection is implemented using simple addition operations in the kernel, ensuring minimal latency by keeping data in the L1 cache or shared memory as much as possible.
2. **skipV1 Block:** Incorporates bottleneck layers in addition to the skip connection, the two convolutional layers in the skipV0 block are replaced by a bottleneck architecture consisting of 1x1 and 3x3 to generate the skipV1 block.The 1x1 convolutions can reduce the feature map's depth before and after the 3x3 convolution, which can be efficiently managed by controlling the threads and memory access patterns. Given the Kepler K40's computing capability, the memory optimization would focus on utilizing shared memory, L2 cache, and registers effectively for these layers to minimize the use of global memory bandwidth.
3. **groupV0 Block:** Uses grouped convolutions, which could be mapped onto different blocks of the GPU to exploit data parallelism. The grouped convolutions can be optimized for memory access by assigning each group to a set of threads within a block, making sure to align the memory access patterns with the warp size.
4. **groupV1 Block:** Combines bottleneck architecture with grouped convolutions, likely using depthwise separable convolutions followed by pointwise convolutions. This structure can reduce parameter count, and for Kepler K40, it would be crucial to optimize the launch configuration and shared memory usage to accommodate the reduced computational demand efficiently.

Grouped convolution blocks, such as "groupV0" and "groupV1," are used in convolutional neural networks (CNNs) to achieve various optimization goals, including reducing the parameter count, improving memory access patterns, and enhancing data parallelism. Here's a closer look at how these blocks work and their role in optimizing CNNs.

Note: A complete block diagram/ flow chart describing our approach has been provided in the Appendix along with the CUDA code mentioning the CUDA libraries used to achieve this.

3. Optimizing CH-CNN using Bayesian Optimization

3.1 Optimizing Parameters

Implementing Bayesian Optimization on the CH-CNN algorithm involves several key steps and considerations. Bayesian Optimization is an efficient method for optimizing complex functions that are expensive to evaluate, such as finding the best hyperparameters for a neural network. In the context of the CH-CNN algorithm, we have chosen to optimize parameters that govern the evolutionary process and the architecture of the CNNs in the initial population stage.

Parameters that can be considered to Optimize:

1. Population Size: The number of CNN architectures in each generation.
2. Number of Generations: The total number of evolutionary cycles.
3. Crossover Rate: The probability of crossover between two CNN architectures.
4. Mutation Rate: The probability of mutations occurring in an individual.
5. Selection Pressure: Governs how the selection is performed, balancing exploration and exploitation.
6. Penalty Coefficients: For penalizing architectures that violate constraints.
7. Repair Probability: The likelihood of applying the repair method to infeasible architectures.
8. CNN Architectural Parameters: This includes the number of layers, types of layers, filter sizes, etc.

Objective Function:

The objective function in Bayesian Optimization for CH-CNN should reflect the goal of finding CNN architectures that are both accurate and comply with resource constraints. A possible formulation could be **Objective function(f) = Accuracy of CNN - λ * Constraint Violation of CNN.**
Here, λ is a factor considering the weight of each constraint.

Surrogate Model:

The surrogate model approximates the objective function based on the observed data points (evaluated CNN architectures). Gaussian Processes are commonly used as surrogate models in Bayesian Optimization due to their ability to estimate uncertainty in predictions, which is crucial for efficient exploration of the parameter space.

We have considered using Bayesian Optimization in finding the initial population metrics that determine the starting point of the genetic algorithm in the CH-CNN. We named the input feature maps for each block as f1, f2, and f3, and the output feature maps from each block are transformed into the next block as input. While considering the pooling layer we have run Bayesian optimization to find the optimal pooling size - 0 to 0.5 for max pooling or 0.5 to 1.0 for the mean pooling. Implementing Bayesian Optimization to determine the optimal population size and number of generations became difficult as it required running the initial CH-CNN on a set of lower to upper bounds for each possibility of population size and number of generations for a predefined set of iterations. The algorithm ran over 4 hours and still does not produce a result as it tries to fit the CH-CNN metrics into the space. Below is a step-by-step implementation that we have chosen while implementing Bayesian optimization to find the optimal hyperparameters for the CH-CNN algorithm. Also, we have used an in-built Bayesian optimization algorithm from the keras

which enabled us to determine the parameter space with lower and upper bounds, activation functions, and optimizers to determine the optimal initial population for the CH-CNN.

3.2 Approach for Optimization

Step-by-step implementation of Bayesian Optimization over CH-CNN:

1. Define the Hyperparameter Space: Determine which hyperparameters of the CH-CNN algorithm you want to optimize. These include:
    a. Depth of the CNN (d).
    b. Parameters related to mutation and crossover operations.(m, c) probabilities.
    c. Number of input feature maps for each block (f1, f2, f3).
    d. Number of epochs in each CNN to determine the batch size to pass.(ep)
    e. Optimizer to use in each CNN block.(op)
    f. Pooling layers count.(p)
    g. Kernel size (k)
2. Define the search space for each hyperparameter, including their minimum and maximum values or allowable ranges.
    a. Depth of the CNN (3,18) - chosen from the paper.
    b. Mutation and Crossover - (0.5, 0.5) - chosen as per the paper.
    c. Input feature maps: f1(32,64) ; f2(64,128); f3(128,512) - chosen from the paper.
    d. Epochs - (0,5) - chosen considering the complexity to run.
    e. Optimizer(op) - ['adamax', 'adadelta', 'adam', 'adagrad'] - as per the guidelines in Keras model of Bayesian Optimization.
    f. Pooling layers count (p) - (0,3) - chosen as per the paper.
    g. Kernel Size (k) - [3,5] means 3*3 / 5*5
3. Define the Objective Function: Define the objective function that you want to optimize using Bayesian optimization. In this case, it would be the performance metric (e.g., accuracy or some other relevant metric) of the CNN architectures generated by CH-CNN on a validation dataset. This function should take the hyperparameters as input and return the performance metric. This is defined as per the equation:
    a. Objective function(f) = Accuracy of CNN - λ * Constraint Violation of CNN.
4. Choose a Surrogate Model: Select a surrogate model for Bayesian optimization. Gaussian Process (GP) regression is a commonly used surrogate model due to its flexibility and ability to model complex objective functions. The inbuilt library of Keras Bayesian Optimization has given the option to use the Gaussian Process to use.
5. Choose an Acquisition Function: Select an acquisition function that guides the optimization process. Common acquisition functions include
    a. Probability of Improvement (PI)
    b. Expected Improvement (EI)
    c. Upper Confidence Bound (UCB)
    d. We have considered Expected improvement in this case as it seems to fit more to the problem of CH-CNN initial population initialization.

6. Initialize Bayesian Optimization: Initialize the Bayesian optimization process with an initial set of hyperparameters. This set is defined in step 2 of this process.
7. Iteratively Optimize: Iteratively perform the following steps until a stopping criterion is met (e.g., a fixed number of iterations or a convergence threshold): We have run the algorithm for iteration steps of 5,10,15,20 to determine and compare the results:
    a. Fit the surrogate model: Use the data collected from previous evaluations of the objective function to fit the surrogate model (GP regression, in this case).
    b. Choose the next hyperparameters to evaluate: Use the acquisition function to select the most promising hyperparameters. These will be passed to the CH-CNN algorithm for evaluation.
    c. Evaluate the objective function: Run the CH-CNN algorithm with the selected hyperparameters, and compute the performance metric on the validation dataset.
    d. Update the surrogate model: Incorporate the new evaluation results into the surrogate model.
8. Stopping Criterion: Define a stopping criterion, such as a maximum number of iterations or a convergence threshold. Bayesian optimization will terminate when this criterion is met. Here it is a set as 5,10,15,20 - run in various stages.
9. Retrieve the Best Configuration: Once the optimization process is complete, retrieve the hyperparameters that correspond to the best-performing configuration of the CH-CNN algorithm.
10. Apply the Best Configuration: Use the best hyperparameters obtained from Bayesian optimization to run the CH-CNN algorithm on your dataset for the final training and evaluation.
11. Analyze Results: Analyze the results of the CH-CNN algorithm with the optimized hyperparameters to assess its performance compared to the initial configurations.

The complete results of the Bayesian Optimization on CH-CNN were discussed in detail in Section 5 of the report. We have compared the results to the initial implementation of the CH-CNN that is optimized for Nvidia Kepler K40 GPU.

4. Optimizing CH-CNN using NSGA-II Algorithm

4.1 Optimizing Parameters

The NSGA-II (Non-dominated Sorting Genetic Algorithm II) is an evolutionary algorithm used for solving multi-objective optimization problems. It's an improvement over its predecessor, NSGA, and is known for its efficiency in handling a large number of objective functions. We can adapt NSGA-II to our CH-CNN algorithm and define multiple objectives to be optimized. The two parameters chosen to be optimized are

1. **Model Accuracy**: This is evident from the evaluate function in the NSGA-II algorithm, where the accuracy of the CNN model is calculated and returned as part of the fitness value. Accuracy is a common and crucial metric in machine learning, especially for classification tasks like those involving the CIFAR-10 dataset. In our code, accuracy is calculated by evaluating the model on the test dataset (test_images and test_labels).
2. **Computational Cost** (as proxied by the number of trainable parameters): The second objective is the computational cost of the CNN model. In our code, this is approximated by the total number of trainable parameters in the model (model.count_params()). This approach assumes that a model with fewer parameters is less computationally intensive, which is generally true. Models with fewer parameters are usually faster to train and infer, and they require less memory, making them more suitable for deployment in environments with limited resources.

These two objectives are typical for optimization in a CNN architecture design. Maximizing accuracy while minimizing the model's complexity (as represented by the number of parameters) is a common goal, as it leads to efficient yet effective models. In the context of NSGA-II, these objectives are simultaneously optimized, providing a set of solutions that represent different trade-offs between accuracy and computational complexity. This allows for a more nuanced selection of models based on specific requirements or constraints, such as resource limitations or the need for high accuracy. We have considered available NSGA II implementations [5],[6] for running the experiments.

4.2 Approach for Optimization

Step-by-step Implementation of integrating NSGA-II algorithm into CH-CNN

1. Modify Fitness Evaluation (In step 3/6 in the CH-CNN): Incorporate multiple objectives into the fitness evaluation:
    a. Objective 1: Accuracy of the CNN.
    b. Objective 2: Complexity or resource usage (like number of parameters).
2. Adapt Population Repair (In step 4/6 in the CH-CNN): Ensure that the repair method considers both objectives. For instance, if a CNN architecture is too complex, the repair method should try to reduce complexity without significantly impacting accuracy.
3. Adjust Offspring Generation (In step 5/6 in the CH-CNN): Continue using crossover and mutation, but ensure these operations respect both objectives. Mutation and crossover should not overly favor one objective over the other.

4. Implement NSGA-II's Selection Mechanism: Replace the environmental selection process with NSGA-II's selection mechanism, which involves:
    a. Non-dominated Sorting: Rank the population into different fronts based on the principle of Pareto dominance. Individuals are sorted into fronts and each front consists of individuals that are not dominated by any member of the previous fronts. This was one of the computationally intensive steps in NSGA, the initial version, and has been updated in NSGA-2.
    b. Crowding Distance Calculation: Within each front, calculate the crowding distance for each individual, which is a measure of how close an individual is to its neighbors. A larger crowding distance means the solution is more diverse.
    c. Selection for Crossover and Mutation: Select individuals based on their rank and crowding distance, prioritizing solutions that are non-dominated and well-distributed.
5. Elite Preservation: Ensure that the best solutions from each front are carried over to the next generation, maintaining a balance between exploration and exploitation.
6. Convergence Criteria: We may need to adjust the convergence criteria or the termination conditions of the algorithm, as multi-objective optimization might require more generations to converge compared to single-objective optimization.

NSGA-II is an effective and efficient method for solving multi-objective optimization problems, offering a good balance between convergence (finding the best solutions) and diversity (exploring various possible solutions). The ability to find a set of optimal solutions (known as the Pareto front) makes it highly suitable for problems where optimizing one objective may worsen others. After the NSGA-II optimization, we will have a Pareto front of non-dominated solutions representing trade-offs between our defined objectives. Analyzing this front to choose the CNN architectures that best suit the requirements based on the objectives we have defined.

5. Results and Analysis

5.1 Experimental Design Setup:

We have a CUDA-written code that depicts the CH-CNN algorithm and provides the best CNN architecture by comparing multiple generations of populations. The code execution has been achieved using Google Colab which supports CUDA Code and Python code execution, and the resource constraints are specified in the code. They do match the free GPU provided for Runtime execution in Colab i.e., 12GB of Onboard RAM in K40 GPU ~ and 15GB of K80 GPU in Google Colab. Also, we have utilized NVCC Plugin [4] that can enable us to run CUDA code in the Colab/ Jupyter environment, and have simultaneous Python code execution as well that we used for Bayesian Optimization and NSGA-II algorithm. All the experiments are run on the CIFAR-10 dataset, which can help us in better comparison of the results. A more detailed cumulative analysis for all the results is provided in Section 5.5.

5.2 Results of the original CH-CNN algorithm without Optimization

We carried out all the experiments for different generations and population scenarios for the initial algorithm run(Bayesian Optimzed CH-CNN and NSGA-II were performed only on Scenario 1):

1. 3 generations - 10 populations in each generation.
2. 3 generations - 15 populations in each generation.
3. 4 generations - 10 populations in each generation.

This would provide us best CNN architecture in each generation and the best architecture overall across the generations. The accuracies are calculated when we evaluate each CNN architecture on the CIFAR-10 dataset and the best accuracy in that population gets to be chosen after repairing all other architectures. Below are the results:

3 Generations - 10 Populations each			
Population	Gen-1 Accuracy	Gen-2 Accuracy	Gen-3 Accuracy
Pop 1	0.734	0.736	0.731
Pop 2	0.746	0.75	0.752
Pop 3	0.691	0.696	0.7
Pop 4	**0.773**	0.663	0.577
Pop 5	0.661	**0.78**	0.66
Pop 6	0.663	0.662	0.725
Pop 7	0.545	0.73	0.745
Pop 8	0.771	0.725	0.745
Pop 9	0.72	0.753	**0.78**
Pop 10	0.721	0.746	0.745

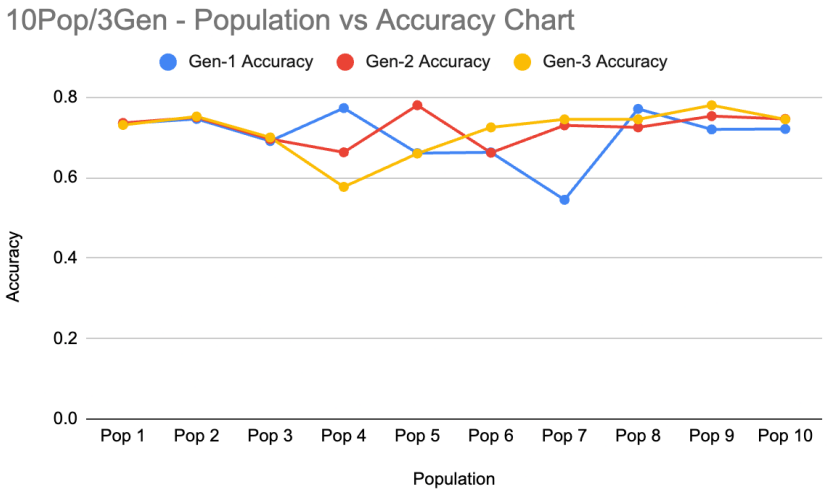
Table 5.1: 3Gen/10Pop Accuracy Chart Table

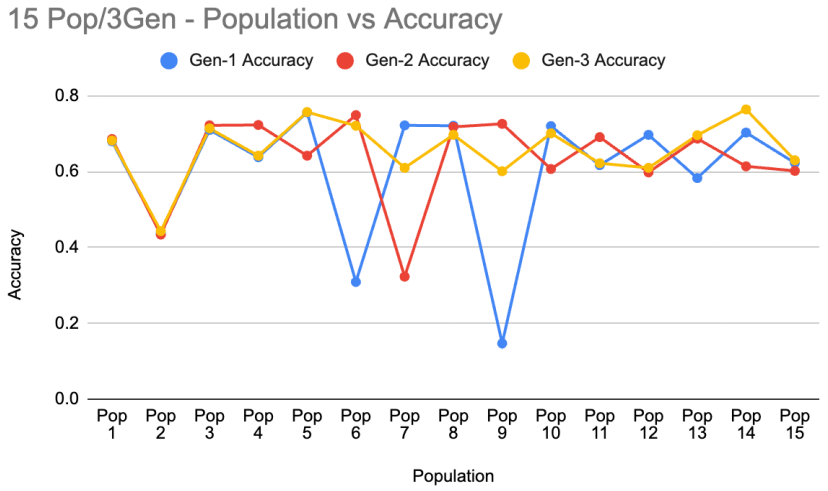
3 Generations - 15 Populations each			
Population	Gen-1 Accuracy	Gen-2 Accuracy	Gen-3 Accuracy
Pop 1	0.68	0.686	0.682
Pop 2	0.437	0.434	0.443
Pop 3	0.71	0.722	0.714
Pop 4	0.638	0.723	0.642
Pop 5	**0.756**	0.642	0.757
Pop 6	0.309	**0.749**	0.721
Pop 7	0.722	0.323	0.61
Pop 8	0.721	0.718	0.697
Pop 9	0.147	0.726	0.601
Pop 10	0.72	0.607	0.701
Pop 11	0.617	0.691	0.622
Pop 12	0.697	0.598	0.61
Pop 13	0.583	0.687	0.696
Pop 14	0.703	0.614	**0.764**
Pop 15	0.623	0.602	0.63

Table 5.2: 3Gen/15Pop Accuracy Chart Table

4 Generations - 10 Populations each				
Population	Gen-1 Accuracy	Gen-2 Accuracy	Gen-3 Accuracy	Gen-4 Accuracy
Pop 1	0.578	0.572	0.5796	0.571
Pop 2	0.652	0.659	0.668	0.682
Pop 3	0.726	0.721	0.728	0.723
Pop 4	0.788	0.79	0.784	0.78
Pop 5	0.72	0.719	0.726	0.711
Pop 6	0.73	0.733	0.728	0.734
Pop 7	0.415	0.627	0.628	0.626
Pop 8	0.775	0.568	0.697	0.698
Pop 9	0.628	0.682	0.537	0.566
Pop 10	0.581	0.695	0.703	0.758

Table 5.3: 3Gen/10Pop Accuracy Chart Table





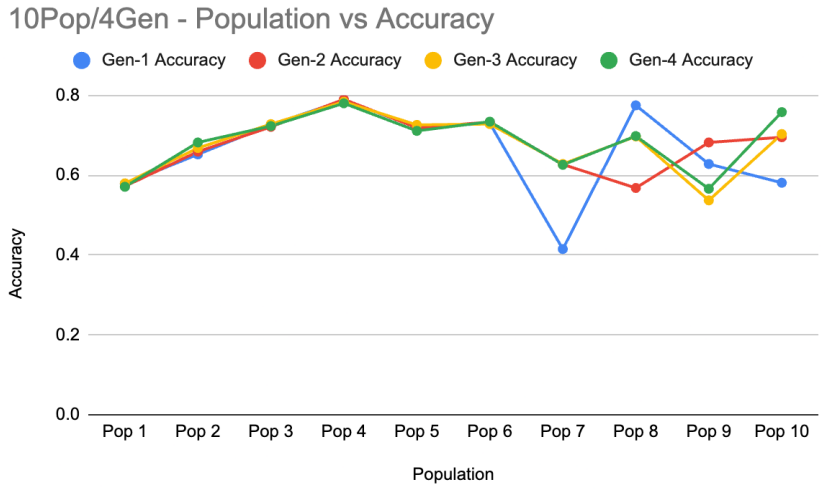


Figure: Population vs Accuracy Charts for each setup

A complete detailed analysis of the tables and charts is provided in Section 5.5. However, other statistics from this experiment are:

1. The runtime is highest for 3gen/15pop type with almost 4hrs of runtime with exceeding RAM usage of 12GB and utilized a disk space of 27GB. This determines that it is very necessary to include high bandwidth and low latency external storage accessible for the microarchitecture hardware to store all the parameters that get generated over the generations.
2. The runtimes for 3gen/10pop are lowest with over 2hrs of runtime and RAM usage of 8GB, and the runtime for 4gen/10pop is around 2hr 30minutes.
3. The decision to not run the experiment further for a huge number of generations is because of limitations in the free tier of Google Colab, we have encountered RAM exceeds issue, Runtime issue as it limits RAM to 12 GB, and a limited set of parallelism libraries to access and also for continuous runtime they window in the browser has to be active until completion if we see the original paper experimental results, where they have used high-end Nvidia GPUs with more memory access and ran the experiment for almost 11 GPU days on CIFAR-10 dataset.

The sample output from the experiment considers the output for generation 3 for 1st setup which achieved an accuracy of 0.78 on the CIFAR-10 dataset.

Output:

Parameters: {'f1': 32, 'f2': 128, 'f3': 256, 'k': 3, 'a1': 'selu', 'a2': 'elu', 'd1': 0.4, 'd2': 0.1, 'op': 'adamax', 'ep': 63}

Accuracy: 0.78

Analysis: It means the first block(f1) has 32 input channels and outputs 128 feature maps, which are fed into the Second block(f2) as 128 input channels and outputs a 256 feature map, and then followed by a pooling layer - (0.4 + 0.1) = 0.5, hence a mean pooling. The kernel size is 'k' = 3 which means a 3*3 convolution kernel is used throughout the architecture of this CNN block. This is considered to avoid the complexity overhead of having multiple convolution kernels. As described in our convolution kernels in Section 2 of the report, storing the convolution calculations in shared memory for a single convolution kernel will be faster and will take less number of records as we consider precomputed values in performing convolution multiplications. Hence this set of CNN architecture models defined with the above parameters gave an accuracy of 0.78 over the CIFAR-10 dataset.

5.3 Results of CH-CNN with Bayesian Optimization

For the Bayesian optimization to initiate, we specified the below bounds (lower, upper) for each parameter that we are trying to optimize and pass these optimized parameters as input to the initial population to the algorithm. The experiment setup remains the same.

Bounds setup for the parameters for Bayesian optimization:

- 'f1': (32, 128), # Example bounds for block 1
- 'f2': (64, 256), #Example bounds for block 2
- 'f3': (128, 512),# block 3
- 'k': (3, 5),# kernel sizes 3 to 5
- 'a1': (0, 2), # Assuming 0 for 'relu', 1 for 'selu', 2 for 'elu' - activation functions
- 'a2': (0, 2), # Activation functions for other blocks
- 'd1': (0.1, 0.5), # Pooling layer
- 'd2': (0.1, 0.5), # Pooling layer
- 'op': (0, 3), # Assuming 0 for 'adamax', 1 for 'adadelta', 2 for 'adam', 3 for 'adagrad' - optimizer

- 'ep': (10, 100) # Number of epochs

The experiment has been run for 5, 10, 15, and 20 iterations in each stage and below are the optimal outputs obtained for each iteration:

Iterations	**Target Accuracy**	**f1**	**f2**	**f3**	**k**	**a1**	**a2**	**d1**	**d2**	**ep**	**op**
5	0.3765	40.86	99.76	260.7	3.794	0.834	1.441	0.1	0.2209	23.21	1.616
10	**0.7193**	**96.36**	**144.1**	**342.5**	**3.281**	**0.7193**	**0.8384**	**1.37**	**0.1818**	**12.46**	**0.5943**
15	0.6759	98.69	137.4	347.1	3.231	1.542	1.961	0.4646	0.3249	17.39	0.1996
20	0.6581	81.23	128.4	287.3	4.322	0.9345	1.2323	0.2234	0.1843	15.43	1.231

We have set the initialization steps to 2 in all the cases. Based on the outputs, we have chosen the parameters defined with iterations of 10 as the initial population setup for the CH-CNN algorithm and ran the experiment again for a population of 3 generations with 10 populations each. The same has been implemented for the NSGA-II algorithm as well. Considering the time taken to run Bayesian Optimization for identifying the optimal hyperparameters for the initial population and again running the CH-CNN algorithm, we have chosen to avoid running for other populations.

3 Generations - 10 Populations each			
Population	Gen-1 Accuracy	Gen-2 Accuracy	Gen-3 Accuracy
Pop 1	0.534	0.526	0.674
Pop 2	0.609	0.638	0.784
Pop 3	0.678	0.674	0.669
Pop 4	**0.779**	0.783	0.537
Pop 5	0.707	**0.815**	0.754
Pop 6	0.533	0.538	0.734
Pop 7	0.443	0.758	0.739
Pop 8	0.594	0.726	0.755
Pop 9	0.772	0.74	**0.771**
Pop 10	0.714	0.748	0.751

Table 5.3: Accuracy Chart Table for 3 generations

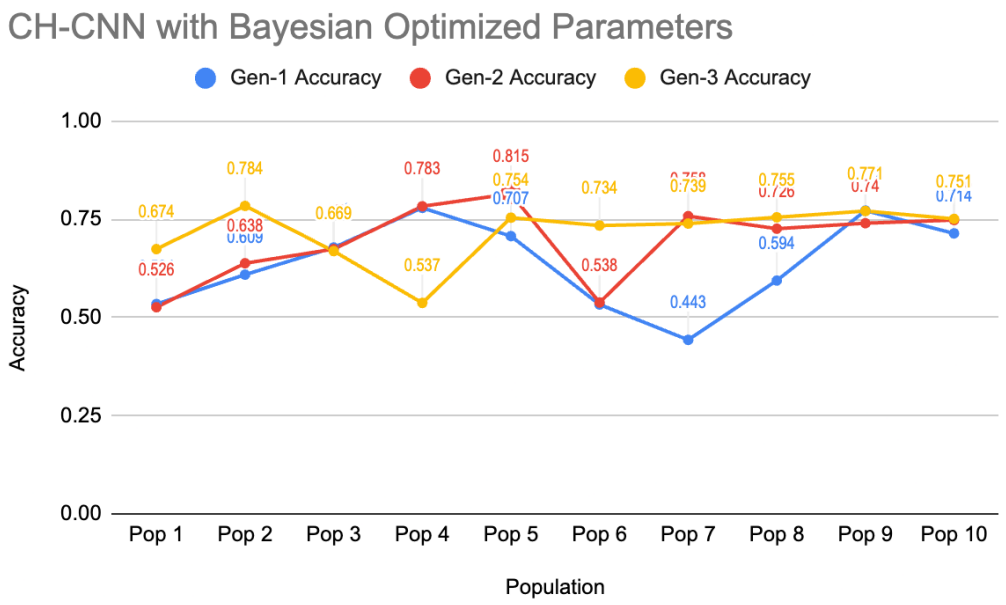


Figure 5.3: Line Chart for Gen-1,2,3 Accuracies

The CH-CNN algorithm when initiated with Bayesian optimized parameters as chosen from above has generated a maximum accuracy of 0.81 for the following CNN architecture parameters:

Parameters: {'k': 3, 'f1': 96, 'f2': 342, 'f3': 512, 'a1': 'selu', 'a2': 'relu', 'd1': 0.1818, 'd2': 0.4512, 'op': 'adamax', 'ep': 12}

Accuracy: 0.815

Analysis: This is very different from the output in Section 5.2, here we have input feature maps in block 1 starting at 96, while it was 32 in the previous case. However this parameter from Bayesian Optimization was obtained by exploring the parameter space that was not considered previously, and there was no CNN architecture previously in any population with a feature maps count reaching 96 in the initial block. This resulted in an accuracy increase of 3%. The experiment is conducted with limitations like the Bayesian Optimization ran for about 2 hours and then the CH-CNN algorithm with Bayesian parameters took another 2hr 30 minutes for the output, a total of almost 5 hours for the execution. The accuracy would have been much better if only many parameters or boundaries had been explored during the Bayesian optimization stage.

5.4 Results of CH-CNN with NSGA-II Algorithm

We have run the NSGA-II algorithm as well for a size of 3 generations with 10 populations each, considering the time taken for each algorithm to execute. Below are the results:

3 Generations - 10 Populations each						
Population	Gen-1 Accuracy	Gen-2 Accuracy	Gen-3 Accuracy	Gen 1 Parameters	Gen 2 Parameters	Gen 3 Parameters
Pop 1	0.323	0.437	0.533	808106	739340	963891
Pop 2	0.718	0.71	0.784	215978	1389092	429834
Pop 3	0.726	0.638	0.61	380618	348239	1782290
Pop 4	0.607	**0.756**	0.697	1838762	592893	342908

Pop 5	0.669	0.309	0.601	674218	230834	1613525
Pop 6	0.537	0.628	0.701	525002	1839022	314255
Pop 7	0.754	0.663	0.622	250154	982489	898235
Pop 8	0.734	**0.791**	0.434	321432	380456	456345
Pop 9	0.538	0.662	0.722	1713409	873593	220119
Pop 10	0.758	0.73	0.723	762481	839211	684012

Table 5.4: Tabulated results of NSGA2 Parameters and Accuracy

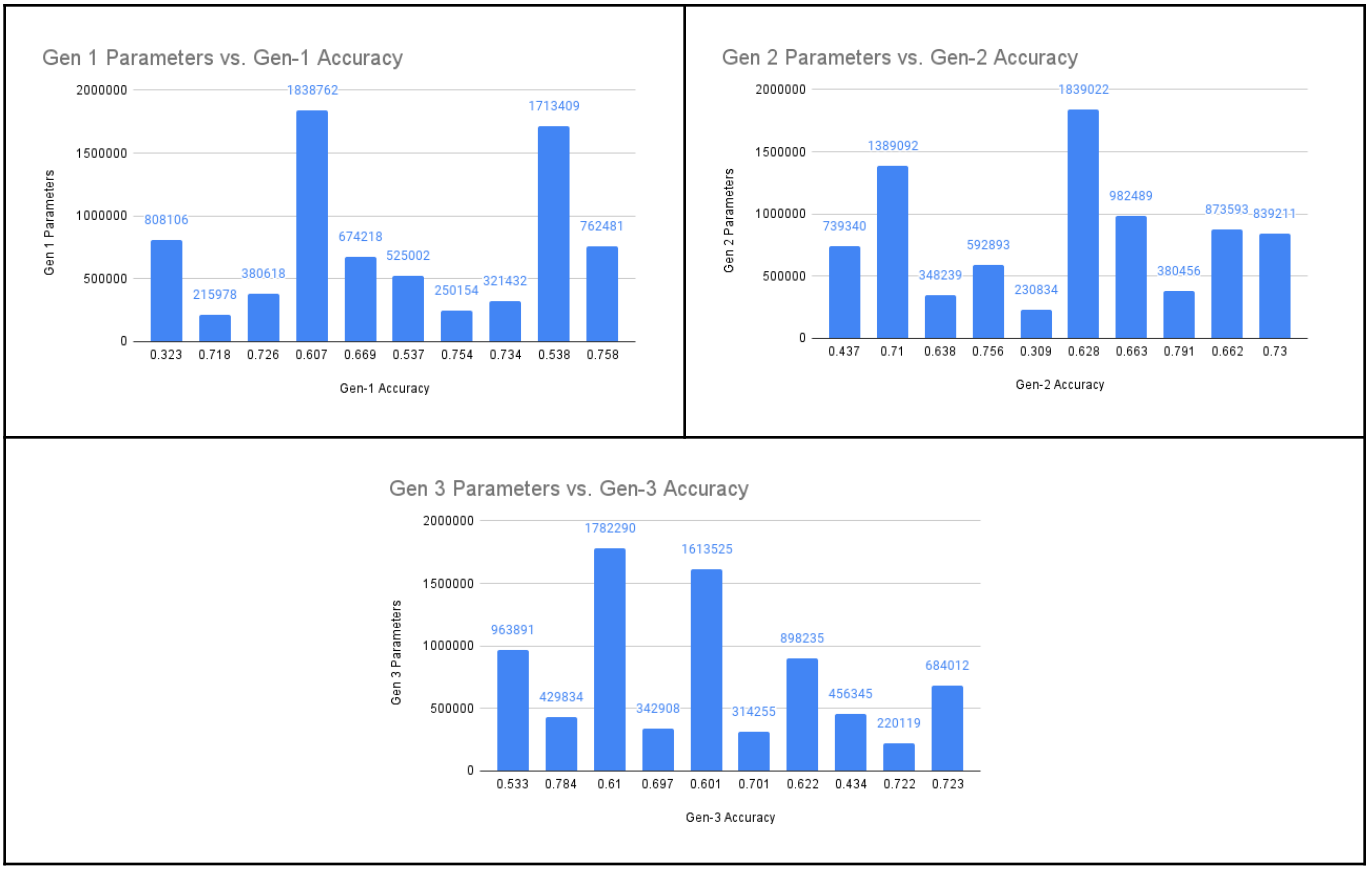


Figure 5.4.1: Parameter vs Accuracy plots for Each Generation

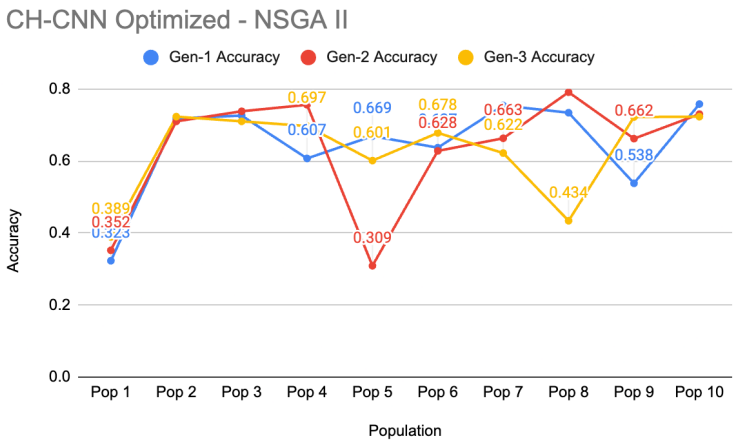


Figure 5.4.2: NSGA2 Accuracy plots in each generation/population

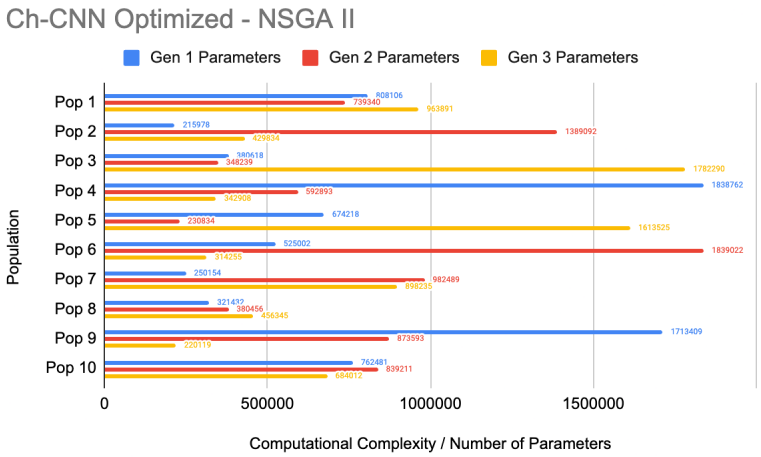


Figure 5.4.3: NSGA2 Parameter plots in each generation/population

5.5 Analysis of the Results

Analysis for Section 5.2 (Original Algorithm from the paper):

1. The algorithm starts each generation space with an almost similar set of parameters and achieves a relatively close accuracy, then deviates for exploration of new generations and then finally follows a similar pattern as other generations at the end.
2. Across all experiments, there is no clear trend of improvement with the increase in generations, which might suggest that the search space is either too vast or that the genetic algorithm parameters need adjustment for better convergence.
3. The algorithm would be performing better if there results start to converge at more population steps would indicate that the genetic algorithm is converging towards certain architectures that perform consistently. This is only seen in the case of the first setup of 3 generations / 10 population.
4. The maximum accuracies reported in each generation were obtained when the algorithm has chosen to explore other parameter combinations i.e., the children/ offspring generated after 3 or 4 CNN architectures in each generation have more mutation and crossover strategies implemented to achieve this.
5. The varied accuracies indicate a good exploration of the architecture space, but they also suggest that there might be a need for a more directed search to improve performance consistently.
6. Outliers: In the case of 3 Generation and 10 populations from Section 5.2, some populations show outlier performances, such as Pop 9 in Gen-1 with a very low accuracy (0.147), which could indicate either an ineffective architecture or a failure in training.

Analysis for Section 5.3 (Bayesian Optimization):

1. Improvement in Accuracy: Compared to previous settings, the optimized parameters led to an accuracy increase of 3%. This highlights the potential of Bayesian optimization in finding effective hyperparameters that were not previously considered.
2. Variability in Results: The genetic algorithm shows variability across generations, with some populations improving and others deteriorating in accuracy. For example, Pop 4 had the highest accuracy in Gen-1 (0.779) but saw a significant drop in Gen-3 (0.537).

3. High Performers: Populations like Pop 5 and Pop 9 maintained high accuracies across generations, indicating the discovery of robust architectures.
4. Low Performers: Some populations like Pop 7 showed low accuracy in the initial generation but improved in subsequent generations, which might be due to the genetic algorithm's exploration and exploitation balance.

Analysis for Section 5.4 (NSGA-II):

1. Trade-off Between Accuracy and Complexity: It is evident from the provided charts and data that there's a trade-off between the number of parameters and the achieved accuracy. NSGA-II seems to be effectively navigating this trade-off, as seen in the performance improvements from Generation 1 to Generation 3 in certain populations.
2. Pareto Optimality: The varied performance across populations with some showing improved accuracy and reduced parameters, while others show the opposite, is characteristic of Pareto optimization where multiple optimal solutions are presented based on varying degrees of trade-offs between objectives.
3. Population Consistency: Some populations like Population 2 consistently performed well across all three generations, which could indicate a robust set of hyperparameters or architecture design that consistently yields good results.
4. Anomalies: Certain anomalies, such as Population 5 in Generation 2, raise questions about the stability of the optimization process across generations. This could be due to the randomness inherent in genetic algorithms or could indicate areas where the algorithm parameters may need adjustment.

Comparison among all three: The original CH-CNN algorithm without optimization seems to have more stable but varied performances compared to the other two methods. In contrast, the Bayesian Optimization and NSGA-II algorithms show greater fluctuations in performance. However, Bayesian Optimization shows some populations with significant improvements over generations, which may indicate its effectiveness in optimizing performance over time. Considering the highest accuracies achieved, the Bayesian Optimization method has the edge with Populations 5 and 2 in Gen-3. The NSGA-II Algorithm results are unique due to the inclusion of parameters. However, no specific insights into how these parameters impact the accuracies is not determined enough. With proper hardware resources if we can run the algorithm for higher generations and populations, we can get deeper insights how the parameters can tune the complexity of the CNN architecture that we design. As a model that processes less parameters and achieving high accuracy will eventually utilise less computational and memory resources, and makes it more suitable for microarchitectures.

6. Conclusion

This report explored the mapping of the CH-CNN algorithm on the NVIDIA Kepler K40 GPU, leveraging its architectural features for optimal performance. The K40's Streaming Multiprocessors (SMX), CUDA cores, and memory hierarchy, including global memory, L1/L2 caches, and registers, were strategically utilized. Dynamic Parallelism and ECC memory support were also key components.

We emphasized GPU memory optimization and efficient data sharing in convolution operations. Parallelism in building blocks was crucial for high-performance CNNs on the GPU. The complete implementation of the CH-CNN algorithm involved population initialization, fitness evaluation, population repair, offspring generation, and environmental selection. An example of CNN architecture demonstrated effective memory usage and parallel processing strategies.

Experiments have been conducted with the initial algorithm and also with optimized versions using Bayesian Optimization and the NSGA-II algorithm on the CIFAR-10 dataset. Bayesian Optimization was employed to optimize key parameters of the CH-CNN algorithm. The approach involved defining the hyperparameter space, and the objective function, selecting a surrogate model, choosing an acquisition function, and iteratively optimizing the hyperparameters. This process aimed to improve the accuracy and efficiency of the CNN architectures. The NSGA-II algorithm was adapted to optimize model accuracy and computational cost. This multi-objective optimization involved modifying fitness evaluation, population repair, and offspring generation processes, implementing NSGA-II's selection mechanism, and analyzing the Pareto front to select optimal CNN architectures.

Experiments conducted on the CIFAR-10 dataset highlighted various findings:

- Original CH-CNN Algorithm: Showed varied accuracies, indicating a need for more directed search and potential for improvement with further generations.
- CH-CNN with Bayesian Optimization: Demonstrated a 3% increase in accuracy, suggesting the effectiveness of Bayesian optimization in discovering efficient hyperparameters.
- CH-CNN with NSGA-II: Revealed a trade-off between accuracy and complexity, with NSGA-II effectively navigating this trade-off.

Future Works:

1. Parameter Tuning: Adjusting the parameters of the genetic algorithm, including mutation rates and crossover strategies, to see if they can guide the search towards more consistently high-performing architectures.
2. Extended Generations: Run experiments for more than 4 generations to observe if there is a longer-term trend toward improvement. This would need hardware that supports its computational requirement of it.
3. Expanding Hyperparameter Space: Future inspections could explore a wider range of hyperparameters in Bayesian Optimization to potentially uncover more efficient CNN architectures.
4. Algorithm Refinement: Refining the genetic algorithm and NSGA-II parameters could lead to more consistent improvements across generations and better stability in the optimization process.

7. References

[1] S. Li, Y. Sun, G. G. Yen and M. Zhang, "Automatic Design of Convolutional Neural Network Architectures Under Resource Constraints," in IEEE Transactions on Neural Networks and Learning Systems, vol. 34, no. 8, pp. 3832-3846, Aug. 2023, doi: 10.1109/TNNLS.2021.3123105.

[2] Chen, X., Chen, J., Chen, D. Z., & Hu, X. S. (2017). Optimizing Memory Efficiency for Convolution Kernels on Kepler GPUs. ArXiv. /abs/1705.10591.

[3] Dong, Shi & Gong, Xiang & Sun, Yifan & Baruah, Trinayan & Kaeli, David. (2018). Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs. 10.1145/3184407.3184423.

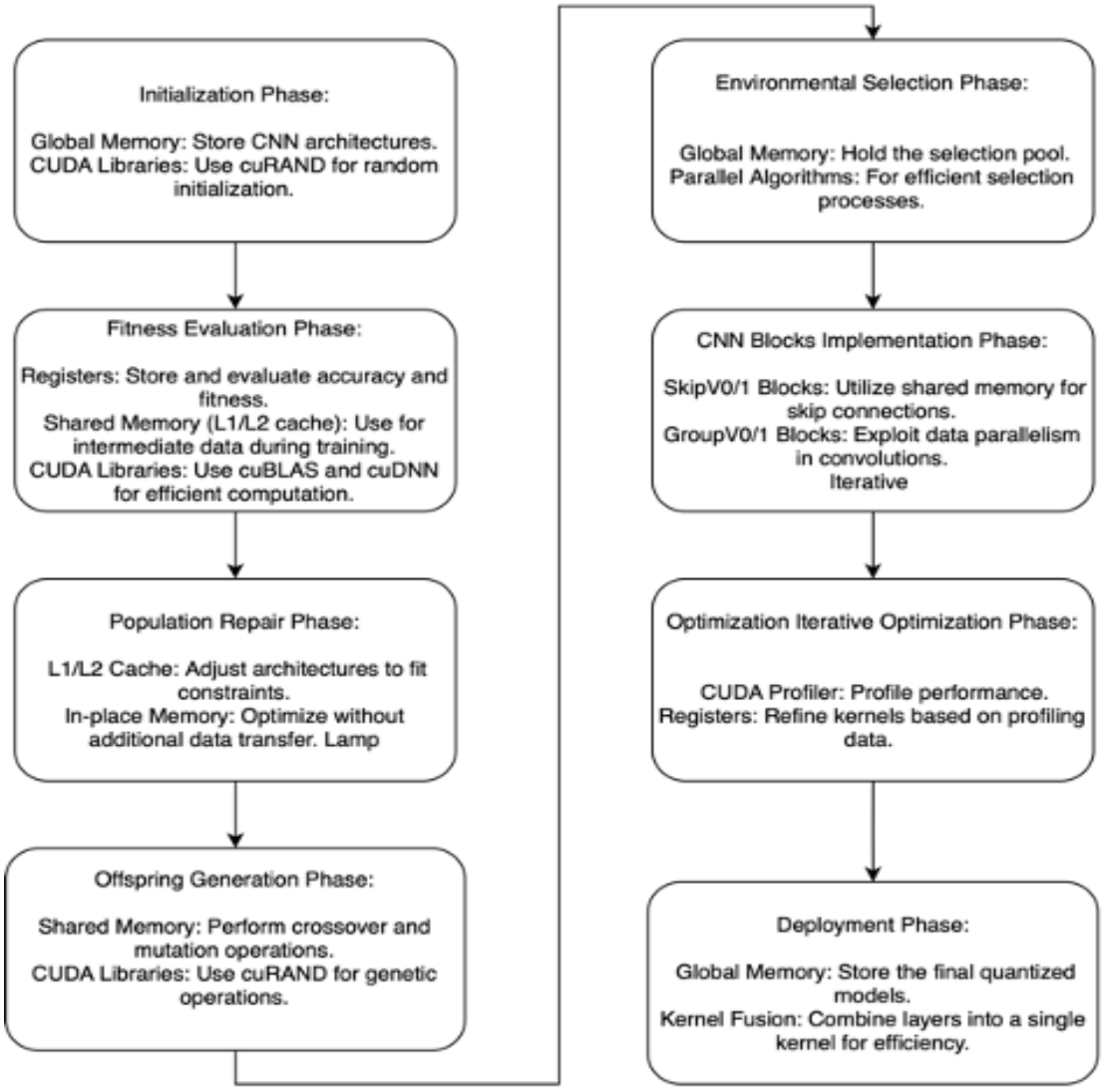
[4] Andreinechaev, A. (2023). nvcc4jupyter. GitHub. https://github.com/andreinechaev/nvcc4jupyter

[5] Harris, H. (2023). NSGA-II: A Python implementation of the Non-dominated Sorting Genetic Algorithm II. GitHub. https://github.com/haris989/NSGA-II

[6] Mikelzc1990. (2023). NSGANetV2: Neural Architecture Search using GANs. GitHub. https://github.com/mikelzc1990/nsganetv2

8. Appendix

A complete flow chart describing our approach :



CUDA Code for CH-CNN:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cuda_runtime.h>
#include <cudnn.h>

// Define constants for image size, kernel size, and other parameters
#define IMAGE_WIDTH 32
#define IMAGE_HEIGHT 32
#define IMAGE_CHANNELS 3
#define KERNEL_SIZE 3
#define NUM_CLASSES 10
#define BATCH_SIZE 100
```

```
#define EPOCHS 50

// Define your CNN model architecture as a sequence of cuDNN layers
void createCNNModel(cudnnHandle_t cudnn, cudnnTensorDescriptor_t input_descriptor,
cudnnTensorDescriptor_t output_descriptor, cudnnConvolutionDescriptor_t
convolution_descriptor, cudnnFilterDescriptor_t filter_descriptor,
cudnnActivationDescriptor_t activation_descriptor, cudnnDropoutDescriptor_t
dropout_descriptor, cudnnRNNDescriptor_t rnn_descriptor) {
     // Initialize and configure cuDNN layers (Convolution, Activation, Pooling, etc.)
here

     cudnnCreateConvolutionDescriptor(&convolution_descriptor);
     cudnnSetConvolution2dDescriptor(convolution_descriptor, ...);
     cudnnCreateFilterDescriptor(&filter_descriptor);
     cudnnSetFilter4dDescriptor(filter_descriptor, ...);

}

// Data preprocessing function
void preprocessData(float* train_images, float* train_labels) {
    // Load and preprocess the dataset here
      train_images =x_train.astype('float32')/255
test_images=x_test.astype('float32')/255
train_labels=to_categorical(y_train)
test_labels=to_categorical(y_test)

}

// Define your training loop using CUDA and cuDNN
void trainCNNModel(float* train_images, float* train_labels, cudnnHandle_t cudnn) {
    // Allocate device memory for images, labels, and model parameters
    // Transfer data from host to device using cudaMemcpy

    // Create cuDNN descriptors and configure the model
    cudnnTensorDescriptor_t input_descriptor, output_descriptor;
    cudnnConvolutionDescriptor_t convolution_descriptor;
    cudnnFilterDescriptor_t filter_descriptor;
    cudnnActivationDescriptor_t activation_descriptor;
    cudnnDropoutDescriptor_t dropout_descriptor;
    cudnnRNNDescriptor_t rnn_descriptor;

     createCNNModel(cudnn, input_descriptor, output_descriptor, convolution_descriptor,
filter_descriptor, activation_descriptor, dropout_descriptor, rnn_descriptor);

    // Training loop
    for (int epoch = 0; epoch < EPOCHS; ++epoch) {
        // Forward and backward passes using cuDNN functions
        // Update model parameters on the GPU
                  cudnnDestroy(cudnn);
               cudaFree(train_images);
               cudaFree(train_labels);
               cudaDeviceReset();
```

```
        // Optionally, perform evaluation on validation set using cuDNN
    }

    // Clean up and release resources
    // Free device memory, destroy cuDNN descriptors, etc.
}

// Mutation function for a chromosome
void mutation(float* chromosome) {
    int flag = rand() % 41;
    if (flag <= 20) {
        // Mutate one or more parameters of the chromosome
            chromosome[0] += some_mutation;
    }
}

// Roulette wheel selection method
void selection(float* population_fitness, int& parent1_ind, int& parent2_ind) {
    // Calculate cumulative probabilities and perform selection
      total = sum(population_fitness)
  percentage = [round((x/total) * 100) for x in population_fitness]
  selection_wheel = []
  for pop_index,num in enumerate(percentage):
    selection_wheel.extend([pop_index]*num)
  parent1_ind = choice(selection_wheel)
  parent2_ind = choice(selection_wheel)
  return [parent1_ind, parent2_ind]

}

// Crossover function for two parents
void crossover(float* parent1, float* parent2, float* child1, float* child2) {
    // Perform crossover between parent1 and parent2 to produce child1 and child2
      child1 = {}
        child2 = {}

        child1["f1"] = choice([parent1["f1"], parent2["f1"]])
        child1["f2"] = choice([parent1["f2"], parent2["f2"]])
        child1["f3"] = choice([parent1["f3"], parent2["f3"]])

        child2["f1"] = choice([parent1["f1"], parent2["f1"]])
        child2["f2"] = choice([parent1["f2"], parent2["f2"]])
        child2["f3"] = choice([parent1["f3"], parent2["f3"]])

        child1["k"] = choice([parent1["k"], parent2["k"]])
        child2["k"] = choice([parent1["k"], parent2["k"]])

        child1["a1"] = parent1["a2"]
        child2["a1"] = parent2["a2"]

        child1["a2"] = parent2["a1"]
        child2["a2"] = parent1["a1"]
```

```
        child1["d1"] = parent1["d1"]
        child2["d1"] = parent2["d1"]

        child1["d2"] = parent2["d2"]
        child2["d2"] = parent1["d2"]

        child1["op"] = parent2["op"]
        child2["op"] = parent1["op"]

        child1["ep"] = parent1["ep"]
        child2["ep"] = parent2["ep"]
        return [child1, child2]

}

// Generate an initial population of chromosomes
void generatePopulation(float* population, int population_size) {
    // Initialize a population of random chromosomes
      population = []
  for i in range(n):
    chromosome = initialization()
    population.append(chromosome)
  return population

}

// Fitness evaluation function
float fitnessEvaluation(float* chromosome) {
    // Evaluate the fitness of a chromosome (e.g., by training a model)
    metrics = model.evaluate(test_images, test_labels)
  computational_cost = model.count_params()
  return metrics[1],computational_cost

}

int main() {
    // Initialize CUDA and cuDNN
    cudnnHandle_t cudnn;
    cudaSetDevice(0);  // Select GPU device if multiple GPUs are available
    cudnnCreate(&cudnn);

    // Allocate memory for training data
    float* train_images;
    float* train_labels;

    // Call the data preprocessing function
    preprocessData(train_images, train_labels);

    // Call the training function with CUDA and cuDNN support
    trainCNNModel(train_images, train_labels, cudnn);

    // Clean up and release resources
```

```
    cudnnDestroy(cudnn);
    cudaFree(train_images);
    cudaFree(train_labels);
    cudaDeviceReset();

    // Genetic Algorithm: Initialize and evolve a population
    int population_size = 10;
    float* population = new float[population_size];
    generatePopulation(population, population_size);

    for (int generation = 0; generation < 3; ++generation) {
        // Evaluate fitness of each chromosome in the population
        float* population_fitness = new float[population_size];
        for (int i = 0; i < population_size; ++i) {
            population_fitness[i] = fitnessEvaluation(&population[i]);
        }

        // Selection, crossover, and mutation steps
        int parent1_ind, parent2_ind;
        selection(population_fitness, parent1_ind, parent2_ind);
        float* parent1 = &population[parent1_ind];
        float* parent2 = &population[parent2_ind];

        float* child1 = new float[NUM_PARAMETERS];
        float* child2 = new float[NUM_PARAMETERS];
        crossover(parent1, parent2, child1, child2);
        mutation(child1);
        mutation(child2);

        // Replace the worst chromosomes with children

        // Clean up resources (free memory, etc.)
        delete[] population_fitness;
        delete[] child1;
        delete[] child2;
    }

    // Clean up resources
    delete[] population;

    return 0;
}
```